下载APP

18 | Python编译器（二）：从AST到字节码

2020-07-13 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 18:31 大小 16.96M

你好，我是宫文学。

今天这一讲，我们继续来研究 Python 的编译器，一起来看看它是如何做语义分析的，以及是如何生成字节码的。学完这一讲以后，你就能回答出下面几个问题了：

> **像 Python 这样的动态语言，在语义分析阶段都要做什么事情呢，跟 Java 这样的静态类型语言有什么不同？**
>
> **Python 的字节码有什么特点？生成字节码的过程跟 Java 有什么不同？**

好了，让我们开始吧。首先，我们来了解一下从 AST 到生成字节码的整个过程。

编译过程

Python 编译器把词法分析和语法分析叫做“**解析（Parse）**”，并且放在 Parser 目录下。而从 AST 到生成字节码的过程，才叫做“**编译（Compile）**”。当然，这里编译的含义是比较狭义的。你要注意，不仅是 Python 编译器，其他编译器也是这样来使用这两个词汇，包括我们已经研究过的 Java 编译器，你要熟悉这两个词汇的用法，以便阅读英文文献。

Python 的编译工作的主干代码是在 Python/compile.c 中，它主要完成 5 项工作。

第一步，检查future 语句。future 语句是 Python 的一个特性，让你可以提前使用未来版本的特性，提前适应语法和语义上的改变。这显然会影响到编译器如何工作。比如，对于“8/7”，用不同版本的语义去处理，得到的结果是不一样的。有的会得到整数“1”，有的会得到浮点数“1.14285...”，编译器内部实际上是调用了不同的除法函数。

第二步，建立符号表。

第三步，为基本块产生指令。

第四步，汇编过程：把所有基本块的代码组装在一起。

第五步，对字节码做窥孔优化。

其中的第一步，它是 Python 语言的一个特性，但不是我们编译技术关注的重点，所以这里略过。我们从建立符号表开始。

语义分析：建立符号表和引用消解

通常来说，在语义分析阶段首先是建立符号表，然后在此基础上做引用消解和类型检查。

而 Python 是动态类型的语言，类型检查应该是不需要了，但引用消解还是要做的。并且你会发现，Python 的引用消解有其独特之处。

首先，我们来看看 Python 的符号表是一个什么样的数据结构。在 **Include/symtable.h** 中定义了两个结构，分别是符号表和符号表的条目：

图 1：符号表和符号表条目

在编译的过程中，针对每个模块（也就是一个 Python 文件）会生成一个**符号表**（symtable）。

Python 程序被划分成“块（block）”，块分为三种：模块、类和函数。每种块其实就是一个作用域，而在 Python 里面还叫做命名空间。**每个块对应一个符号表条目（PySTEntryObject），每个符号表条目里存有该块里的所有符号（ste_symbols）。**每个块还可以有多个子块（ste_children），构成树状结构。

在符号表里，有一个 st_blocks 字段，这是个字典，它能通过模块、类和函数的 AST 节点，查找到 Python 程序的符号表条目，通过这种方式，就把 AST 和符号表关联在了一起。

我们来看看，对于下面的示例程序，它对应的符号表是什么样子的。

复制代码

```
a = 2 #模块级变量
class myclass:
    def __init__(self, x):
        self.x = x
    def foo(self, b):
        c = a + self.x + b  #引用了外部变量a
        return c
```

这个示例程序有模块、类和函数三个级别的块。它们分别对应一条符号表条目。

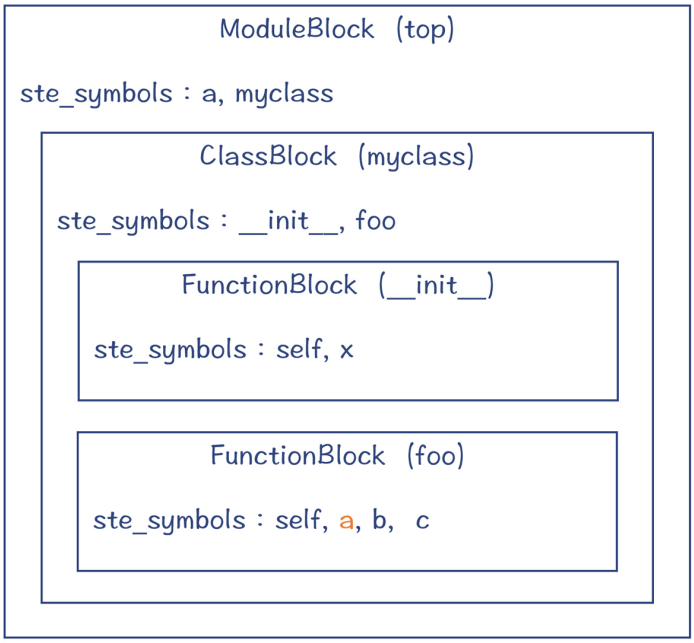


图 2：示例程序对应的符号表

你可以看到，每个块里都有 ste_symbols 字段，它是一个字典，里面保存了本命名空间涉及的符号，以及每个符号的各种标志位（flags）。关于标志位，我下面会给你解释。

然后，我们再看看针对这个示例程序，符号表里的主要字段的取值：

字段	取值	解释
st_cur	foo块	当前正在处理的块是foo
st_top	top块	顶层的块，也就是模块所对应的块
st_blocks	模块节点->top myclass节点->myclass块 __init__节点->__init__块 foo节->foo块	一个字典，可以通过AST节点找到它对应的符号表中的块
st_stack	top, myclass, foo	命名空间的栈 通过这个栈可以查找上一级命名空间中的符号

好了，通过这样一个例子，你大概就知道了 Python 的符号表是怎样设计的了。下面我们来看看符号表的建立过程。

建立符号表的主程序是 Python/symtable.c 中的 **PySymtable_BuildObject() 函数**。

Python 建立符号表的过程，需要做两遍处理，如下图所示。

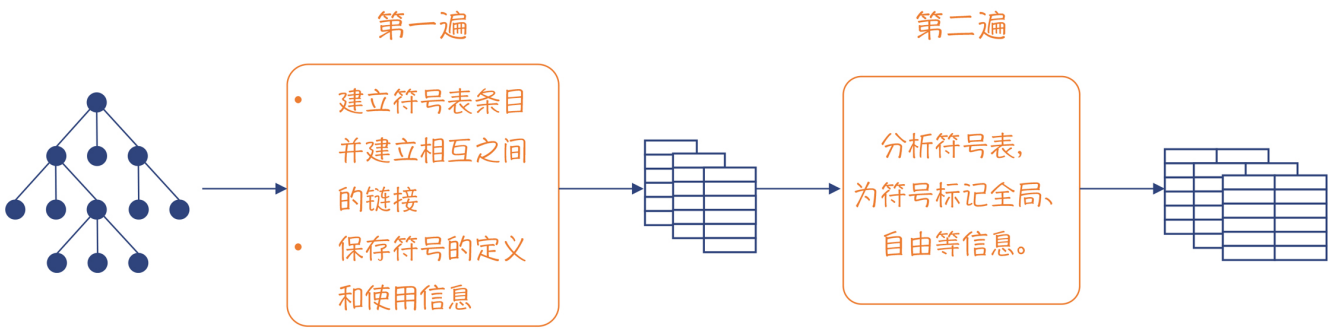


图 3：Python 建立符号表的过程

第一遍，主要做了两件事情。第一件事情是建立一个个的块（也就是符号表条目），并形成树状结构，就像示例程序那样；第二件事情，就是给块中的符号打上一定的标记（flag）。

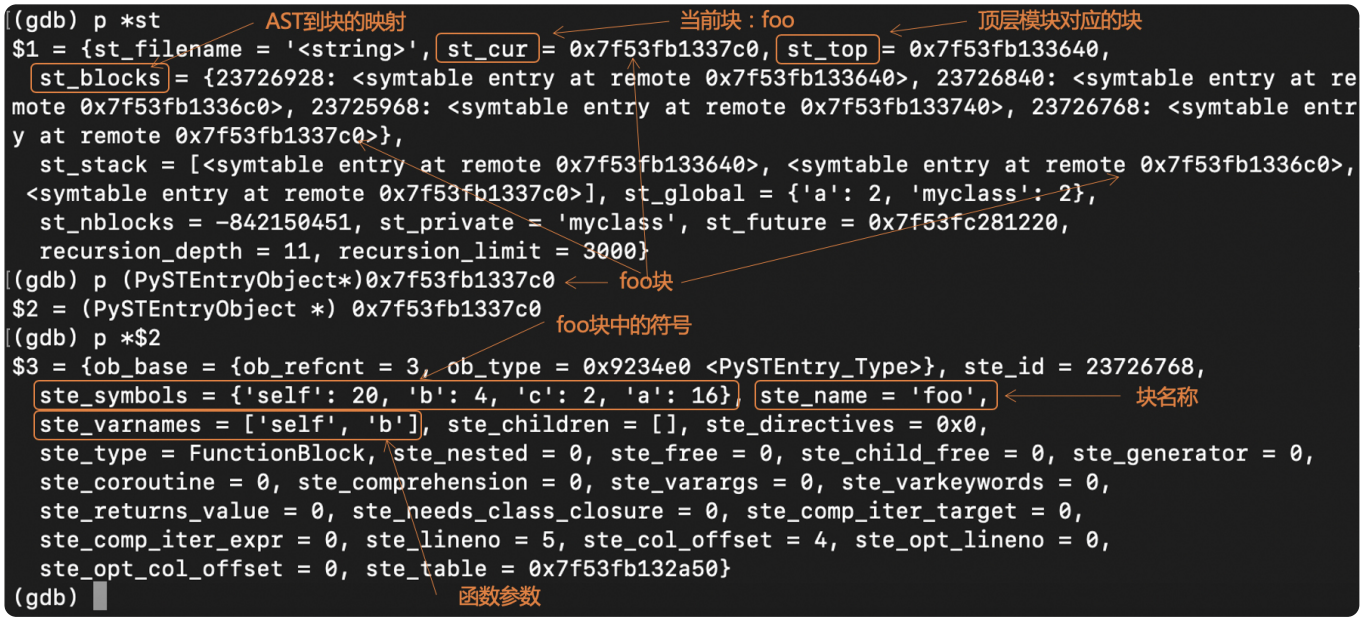
我们用 GDB 跟踪一下第一遍处理后生成的结果。你可以参考下图，看一下我在 Python 的 REPL 中的输入信息：

```
>>> s = """a = 2 #模块级变量
class myclass:
    def __init__(self, x):
        self.x = x
    def foo(self, b):
        c = a + self.x + b  #引用了外部变量a
        return c... ... ... ... ... ...
... """
>>> exec(s)
```

把示例程序保存到字符串s中

执行示例程序，会编译，并触发断点

我在 symtable_add_def_helper() 函数中设置了断点，便于调试。当编译器处理到 foo 函数的时候，我在 GDB 中打印输出了一些信息：




```
(gdb) p *st
$1 = {st_filename = '<string>', st_cur = 0x7f53fb1337c0, st_top = 0x7f53fb133640,
  st_blocks = {23726928: <symtable entry at remote 0x7f53fb133640>, 23726840: <symtable entry at re
mote 0x7f53fb1336c0>, 23725968: <symtable entry at remote 0x7f53fb133740>, 23726768: <symtable entr
y at remote 0x7f53fb1337c0>},
  st_stack = [<symtable entry at remote 0x7f53fb133640>, <symtable entry at remote 0x7f53fb1336c0>,
 <symtable entry at remote 0x7f53fb1337c0>], st_global = {'a': 2, 'myclass': 2},
  st_nblocks = -842150451, st_private = 'myclass', st_future = 0x7f53fc281220,
  recursion_depth = 11, recursion_limit = 3000}
(gdb) p (PySTEntryObject*)0x7f53fb1337c0
$2 = (PySTEntryObject *) 0x7f53fb1337c0
(gdb) p *$2
$3 = {ob_base = {ob_refcnt = 3, ob_type = 0x9234e0 <PySTEntry_Type>}, ste_id = 23726768,
  ste_symbols = {'self': 20, 'b': 4, 'c': 2, 'a': 16}, ste_name = 'foo',
  ste_varnames = ['self', 'b'], ste_children = [], ste_directives = 0x0,
  ste_type = FunctionBlock, ste_nested = 0, ste_free = 0, ste_child_free = 0, ste_generator = 0,
  ste_coroutine = 0, ste_comprehension = 0, ste_varargs = 0, ste_varkeywords = 0,
  ste_returns_value = 0, ste_needs_class_closure = 0, ste_comp_iter_target = 0,
  ste_comp_iter_expr = 0, ste_lineno = 5, ste_col_offset = 4, ste_opt_lineno = 0,
  ste_opt_col_offset = 0, ste_table = 0x7f53fb132a50}
(gdb)
```

在这些输出信息中，你能看到前面我给你整理的表格中的信息，比如，符号表中各个字段的取值。

我重点想让你看的，是 foo 块中各个符号的标志信息：self 和 b 是 20，c 是 2，a 是 16。这是什么意思呢？

复制代码

```
ste_symbols = {'self': 20, 'b': 20, 'c': 2, 'a': 16}
```

这就需要看一下 symtable.h 中，对这些标志位的定义：

```
/* Flags for def-use information */

#define DEF_GLOBAL 1           /* global stmt */
#define DEF_LOCAL 2            /* assignment in code block */
#define DEF_PARAM 2<<1         /* formal parameter */
#define DEF_NONLOCAL 2<<2      /* nonlocal stmt */
#define USE 2<<3               /* name is used */
#define DEF_FREE 2<<4          /* name used but not defined in nested block */
#define DEF_FREE_CLASS 2<<5    /* free variable from class's method */
#define DEF_IMPORT 2<<6        /* assignment occurred via import */
#define DEF_ANNOT 2<<7         /* this name is annotated */
#define DEF_COMP_ITER 2<<8     /* this name is a comprehension iteration variable */
```

我给你整理成了一张更容易理解的图，你参考一下：

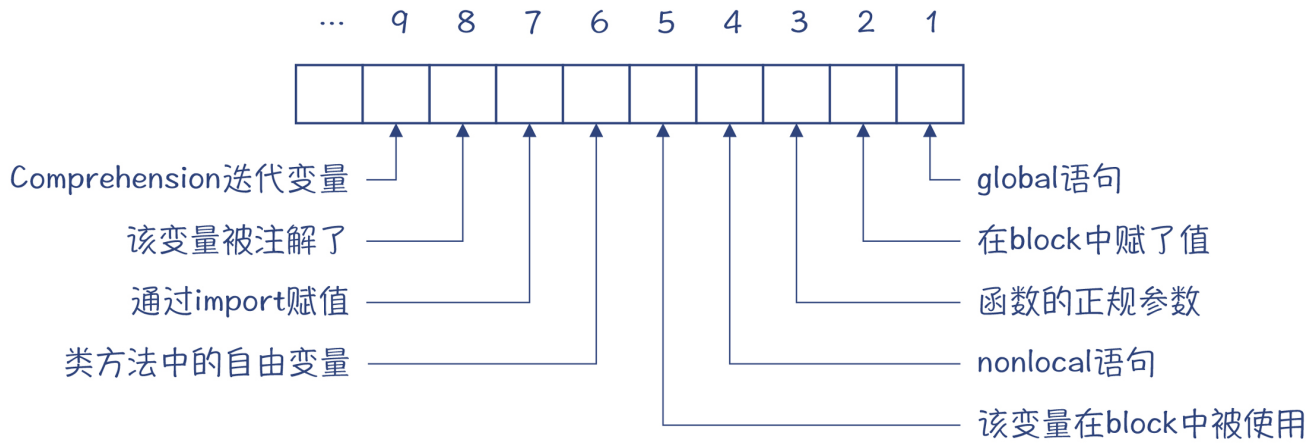


图 4：符号标志信息中每个位的含义

根据上述信息，你会发现 self 和 b，其实是被标记了 3 号位和 5 号位，意思是这两个变量是函数参数，并且在 foo 中被使用。而 a 只标记了 5 号位，意思是 a 这个变量在 foo 中被使用，但这个变量又不是参数，所以肯定是来自外部作用域的。我们再看看 c，c 只在 2 号位被标记，表示这个变量在 foo 里被赋值了。

到目前为止，第一遍处理就完成了。通过第一遍处理，我们会知道哪些变量是本地声明的变量、哪些变量在本块中被使用、哪几个变量是函数参数等几方面的信息。

但是，现在还有一些信息是不清楚的。比如，在 foo 中使用了 a，那么外部作用域中是否有这个变量呢？这需要结合上下文做一下分析。

还有，变量 c 是在 foo 中赋值的。那它是本地变量，还是外部变量呢？

在这里，你能体会出 **Python 语言使用变量的特点：由于变量在赋值前，可以不用显式声明。**所以你不知道这是新声明了一个变量，还是引用了外部变量。

正由于 Python 的这个特点，所以它在变量引用上有一些特殊的规定。

比如，想要在函数中给全局变量赋值，就必须加 **global 关键字**，否则编译器就会认为这个变量只是一个本地变量。编译器会给这个符号的 1 号位做标记。

而如果给其他外部作用域中的变量赋值，那就必须加 **nonlocal 关键字**，并在 3 号位上做标记。这时候，该变量就是一个自由变量。在闭包功能中，编译器还要对自由变量的存储

做特殊的管理。

接下来，编译器会**做第二遍的分析**（见 symtable_analyze() 函数）。在这遍分析里，编译器会根据我们刚才说的 Python 关于变量引用的语义规则，分析出哪些是全局变量、哪些是自由变量，等等。这些信息也会被放到符号的标志位的第 12~15 位。

```
1 ste_symbols = {'self': 2068, 'b': 2068, 'c': 2050, 'a': 6160}
```

复制代码

```
/* GLOBAL_EXPLICIT and GLOBAL_IMPLICIT are used internally by the symbol
   table.  GLOBAL is returned from PyST_GetScope() for either of them.
   It is stored in ste_symbols at bits 12-15.
*/
#define SCOPE_OFFSET 11
#define SCOPE_MASK (DEF_GLOBAL | DEF_LOCAL | DEF_PARAM | DEF_NONLOCAL)

#define LOCAL 1
#define GLOBAL_EXPLICIT 2
#define GLOBAL_IMPLICIT 3
#define FREE 4
#define CELL 5
```

图 5：symtable.h 中对作用域的标志位

以变量 a 为例，它的标志值是 6160，也就是二进制的 1100000010000。其标记位设置如下，其作用域的标志位是 3，也就是说，a 是个隐式的全局变量。而 self、b 和 c 的作用域标志位都是 1，它们的意思是本地变量。

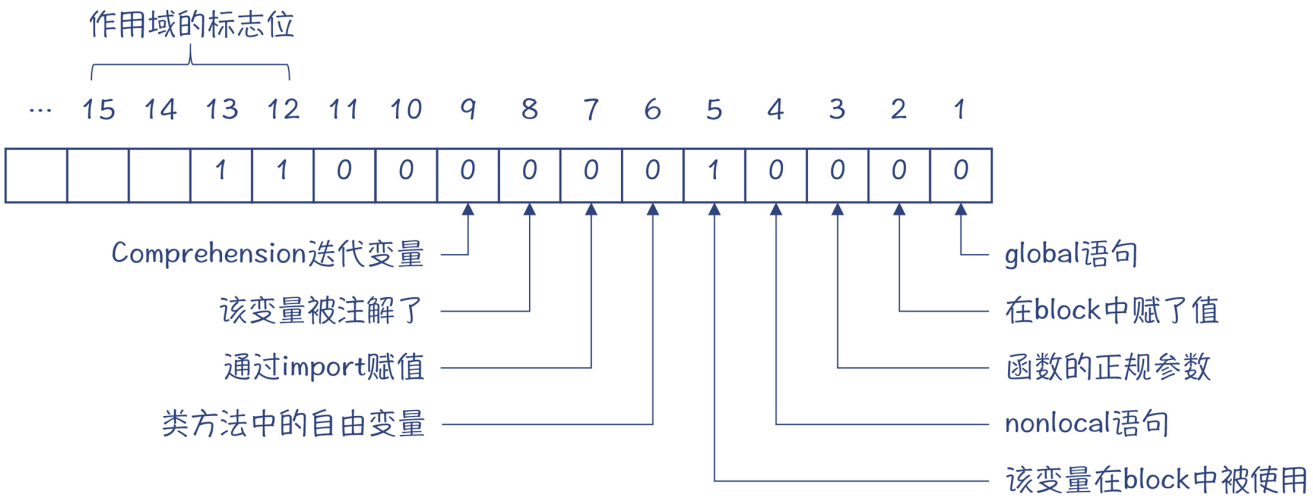


图 6：作用域的标志位

在第二遍的分析过程中，Python 也做了一些语义检查。你可以搜索一下 Python/symtable.c 的代码，里面有很多地方会产生错误信息，比如“nonlocal declaration not allowed at module level（在模块级不允许非本地声明）”。

另外，Python 语言提供了访问符号表的 API，方便你直接在 REPL 中，来查看编译过程中生成的符号表。你可以参考我的屏幕截图：

```
>>> import symtable
>>> s = """
... a = 2 #模块级变量
... class myclass:
...     def __init__(self, x):
...         self.x = x
...     def foo(self, b):
...         c = a + self.x + b  #引用了外部变量a
...         return c
... """
>>> table = symtable.symtable(s,"string","exec")
>>> table.get_symbols()
[<symbol 'a'>, <symbol 'myclass'>]
>>> symmyclass = table.lookup('myclass')
>>> type(symmyclass)
<class 'symtable.Symbol'>
>>> nsmyclass = symmyclass.get_namespace()
>>> type(nsmyclass)
<class 'symtable.Class'>
>>> nsmyclass.get_symbols()
[<symbol '__init__'>, <symbol 'foo'>]
>>> nsfoo = nsmyclass.lookup('foo').get_namespace()
>>> nsfoo.get_symbols()
[<symbol 'self'>, <symbol 'b'>, <symbol 'c'>, <symbol 'a'>]
>>> nsfoo.get_globals()
('a',)
>>> nsfoo.get_parameters()
('self', 'b')
>>>
```


好了，现在符号表已经生成了。基于 AST 和符号表，Python 编译器就可以生成字节码。

生成 CFG 和指令

我们可以用 Python 调用编译器的 API，来观察字节码生成的情况：

```
>>> co = compile("a+2", "test.py", "eval")  //编译表达式"a+2"
>>> dis.dis(co.co_code)                     //反编译字节码
           0 LOAD_NAME                0 (0)  //装载变量a
           2 LOAD_CONST               0 (0)  //装载常数2
           4 BINARY_ADD                      //执行加法
           6 RETURN_VALUE                    //返回值
```

其中的 LOAD_NAME、LOAD_CONST、BINARY_ADD 和 RETURN_VALUE 都是字节码的指令。

对比一下，Java 的字节码的每个指令只有一个字节长，这意味着指令的数量不会超过 2 的 8 次方（256）个。

Python 的指令一开始也是一个字节长的，后来变成了一个字（word）的长度，但我们仍然习惯上称为字节码。Python 的在线文档里有对所有字节码的说明，这里我就不展开了，感兴趣的话你可以自己去看看。

并且，Python 和 Java 的虚拟机一样，都是基于栈的虚拟机。所以，它们的指令也很相似。比如，加法操作的指令是不需要带操作数的，因为只需要取出栈顶的两个元素相加，把结果再放回栈顶就行了。

进一步，你可以对比一下这两种语言的字节码，来看看它们的异同点，并且试着分析一下原因。

	Python指令	Java指令
加载数据到栈顶	LOAD_NAME namei LOAD_CONST consti LOAD_GLOBAL namei LOAD_FAST var_num	iconst_x //较短的整型常数 aload index //引用 iload index //整型 lload index //长整型 fload index //单精度浮点数 dload index //双精度浮点数 ldc //较长的常数
把栈里的数据保存到变量	STORE_NAME namei STORE_GLOBAL namei STORE_FAST var_num	astore index istrore index fstore index dstore index
加法（栈顶两个元素相加）	BINARY_ADD	iadd ladd fadd dadd
返回数据到函数的调用者	RETURN_VALUE	ireturn lreturn freturn dreturn
重复栈顶元素	DUP_TOP	dup
弹出栈顶元素	POP_TOP	pop

这样对比起来，你可以发现，它们主要的区别就在于，**Java 的字节码对不同的数据类型会提供不同的指令，而 Python 则不加区分。**因为 Python 对所有的数值，都会提供统一的计算方式。

所以你可以看出，一门语言的 IR，是跟这门语言的设计密切相关的。

生成 CFG 和字节码的代码在 **Python/compile.c** 中。调用顺序如下：

程序文件	函数	说明
...	...	...
Python/compiler.c	PyAST_CompileObject()	把AST编译成字节码
Python/compiler.c	compiler_mod()	编译模块
Python/compiler.c	compiler_body()	编译模块的主体部分
Python/compiler.c	compiler_visit_stmt() compiler_visit_expr() compiler_visit_expr() compiler_for() ...	用visit模式遍历AST，生成CFG和字节码的指令。

Python/compiler.c	compiler_addop() //无参数
	compiler_addop_i() //1个整型参数
	compiler_addop_o() //1个对象参数
		往compiler结构中增加指令。

总的逻辑是：以 visit 模式遍历整个 AST，并建立基本块和指令。对于每种 AST 节点，都由相应的函数来处理。

以 compiler_visit_expr1() 为例，对于二元操作，编译器首先会递归地遍历左侧子树和右侧子树，然后根据结果添加字节码的指令。

复制代码

```
compiler_visit_expr1(struct compiler *c, expr_ty e)
{
    switch (e->kind) {
...
.
    case BinOp_kind:
        VISIT(c, expr, e->v.BinOp.left);     //遍历左侧子树
        VISIT(c, expr, e->v.BinOp.right);    //遍历右侧子树
        ADDOP(c, binop(c, e->v.BinOp.op));   //添加二元操作的指令
        break;
...
}
```

那么基本块是如何生成的呢？

编译器在进入一个作用域的时候（比如函数），至少要生成一个基本块。而像循环语句、if 语句，还会产生额外的基本块。

所以，编译的结果，会在 compiler 结构中保存一系列的基本块，这些基本块相互连接，构成 CFG；基本块中又包含一个指令数组，每个指令又包含操作码、参数等信息。

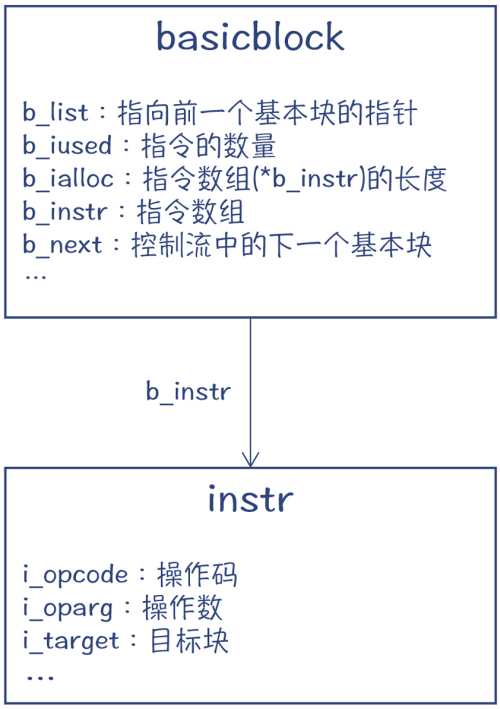


图 7：基本块和指令

为了直观理解，我设计了一个简单的示例程序。foo 函数里面有一个 if 语句，这样会产生多个基本块。

复制代码

```
1 def foo(a):
2     if a > 10 :
3         b = a
4     else:
5         b = 10
6     return b
```

通过 GDB 跟踪编译过程，我们发现，它生成的 CFG 如下图所示：

图 8：示例程序对应的 CFG

在 CFG 里，你要注意两组箭头。

实线箭头是基本块之间的跳转关系，用 b_next 字段来标记。**虚线箭头**能够基于 b_list 字段把所有的基本块串起来，形成一个链表，每一个新生成的基本块指向前一个基本块。只要有一个指针指向最后一个基本块，就能访问所有的基本块。

你还可以通过 GDB 来查看每个基本块中的指令分别是什么，这样你就会理解每个基本块到底干了啥。不过我这里先给你留个小伏笔，在下一个小节讲汇编的时候一起给你介绍。

到目前为止，我们已经生成了 CFG 和针对每个基本块的指令数组。但我们还没有生成最后的字节码。这个任务，是由汇编（Assembly）阶段负责的。

汇编（Assembly）

汇编过程是在 Python/compiler.c 中的 assemble() 函数中完成的。听名字，你会感觉这个阶段做的事情似乎应该比较像汇编语言的汇编器的功能。也确实如此。汇编语言的汇编器，能够生成机器码；而 Python 的汇编阶段，是生成字节码，它们都是生成目标代码。

具体来说，汇编阶段主要会完成以下任务：

把每个基本块的指令对象转化成字节码。

把所有基本块的字节码拼成一个整体。

对于从一个基本块跳转到另一个基本块的 jump 指令，它们有些采用的是相对定位方式，比如往前跳几个字的距离。这个时候，编译器要计算出正确的偏移值。

生成 PyCodeObject 对象，这个对象里保存着最后生成的字节码和其他辅助信息，用于 Python 的解释器执行。

我们还是通过示例程序，来直观地看一下汇编阶段的工作成果。你可以参照下图，使用 instaviz 工具看一下 foo 函数的编译结果。

```
>>> import instaviz
>>> def foo(a):
...     if a > 10 :
...         b = a
...     else:
...         b = 10
...     return b
...
>>> instaviz.show(foo)
Bottle v0.12.18 server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

在 PyCodeObject 对象中，co_code 字段是生成的字节码（用 16 进制显示）。你还能看到常量表和变量表，这些都是在解释器中运行所需要的信息。

Field	Value
co_argcount	1
co_cellvars	()
co_code	7c0064016b04720e7c007d016e0464017d017c015300
co_consts	(None, 10)
co_filename	
co_firstlineno	1
co_flags	67
co_freevars	()
co_kwonlyargcount	0
co_lnotab	b'\x00\x01\x08\x01\x06\x02\x04\x01'
co_name	foo
co_names	()
co_nlocals	2
co_posonlyargcount	0
co_stacksize	2
co_varnames	('a', 'b')

如果把 co_code 字段的那一串字节码反编译一下，你会得到下面的信息：

Disassembled Code

	OpCode	Operation Name	Numeric Arg	Resolved Arg Value	Argument description	Index Offset	Starts Line	Is Jump Target?
BB1	124	LOAD_FAST	0	a	a	0	2	False
	100	LOAD_CONST	1	10	10	2	None	False
	107	COMPARE_OP	4	>	>	4	None	False
	114	POP_JUMP_IF_FALSE	14	14		6	None	False
	124	LOAD_FAST	0	a	a	8	3	False
	125	STORE_FAST	1	b	b	10	None	False
	110	JUMP_FORWARD	4	18	to 18	12	None	False
BB2	100	LOAD_CONST	1	10	10	14	5	True
	125	STORE_FAST	1	b	b	16	None	False
BB3	124	LOAD_FAST	1	b	b	18	6	True
	83	RETURN_VALUE	None	None		20	None	False

计算出正确的偏移量

你会看到，一共 11 条指令，其中 BB1 是 7 条，BB2 和 BB3 各 2 条。BB1 里面是 If 条件和 if 块中的内容，BB2 对应的是 else 块的内容，BB3 则对应 return 语句。

不过，如果你对照基本块的定义，你其实会发现，BB1 不是一个标准的基本块。因为一般来说，标准的基本块只允许在最后一个语句做跳转，其他语句都是顺序执行的。而我们看到第 4 个指令“POP_JUMP_IF_FALSE 14”其实是一个条件跳转指令。

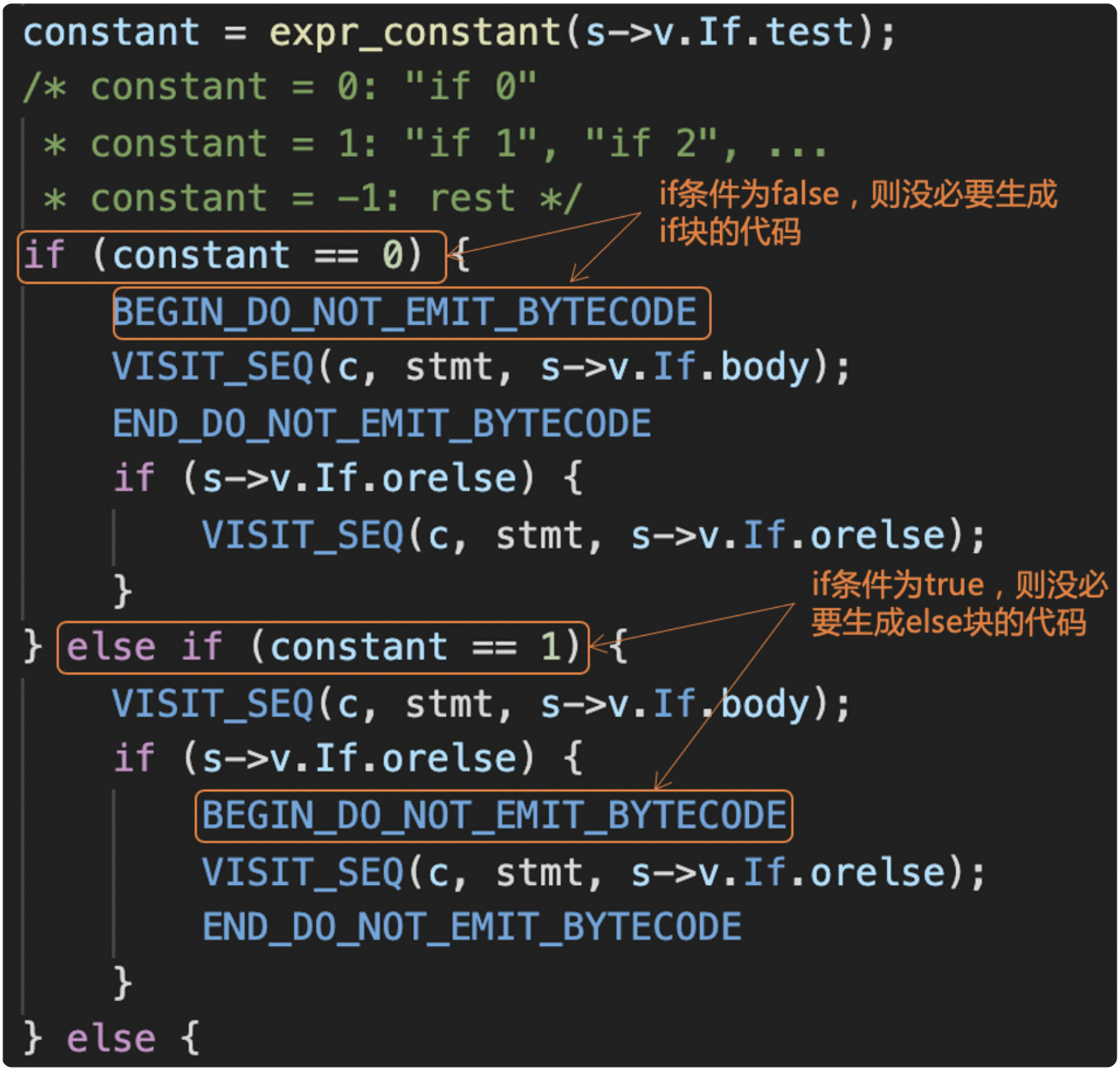
尽管如此，因为 Python 把 CFG 只是作为生成字节码的一个中间结构，并没有基于 CFG 做数据流分析和优化，所以虽然基本块不标准，但是对 Python 的编译过程并无影响。

你还会注意到第 7 行指令“JUMP_FORWARD”，这个指令是一个基于相对位置的跳转指令，它往前跳 4 个字，就会跳到 BB3。这个跳转距离就是在 assemble 阶段去计算的，这个距离取决于你如何在代码里排列各个基本块。

好了，到目前为止，字节码已经生成了。不过，在最后放到解释器里执行之前，编译器还会再做一步窥孔优化工作。

窥孔优化

说到优化，总体来说，在编译的过程中，Python 编译器的优化功能是很有限的。在 compiler.c 的代码中，你会看到一点优化逻辑。比如，在为 if 语句生成指令的时候，编译器就会看看 if 条件是否是个常数，从而不必生成 if 块或者 else 块的代码。



另一个优化机会，就是在字节码的基础上优化，这就是窥孔优化，其实现是在 **Python/peephole.c** 中。它能完成的优化包括：

把多个 LOAD_CONST 指令替换为一条加载常数元组的指令。

如果一个跳转指令，跳到 return 指令，那么可以把跳转指令直接替换成 return 指令。

如果一个条件跳转指令，跳到另一个条件跳转指令，则可以基于逻辑运算的规则做优化。比如，“x:JUMP_IF_FALSE_OR_POP y”和“y:JUMP_IF_FALSE_OR_POP z”可以直接简化为“x:JUMP_IF_FALSE_OR_POP z”。这是什么意思呢？第一句是依据栈顶的值做判断，如果为 false 就跳转到 y。而第二句，继续依据栈顶的值做判断，如果为 false 就跳转到 z。那么，简化后，可以直接从第一句跳转到 z。

去掉 return 指令后面的代码。

……

在做优化的时候，窥孔优化器会去掉原来的指令，替换成新的指令。如果有多余出来的位置，则会先填充 NOP 指令，也就是不做任何操作。最后，才扫描一遍整个字节码，把 NOP 指令去掉，并且调整受影响的 jump 指令的参数。

课程小结

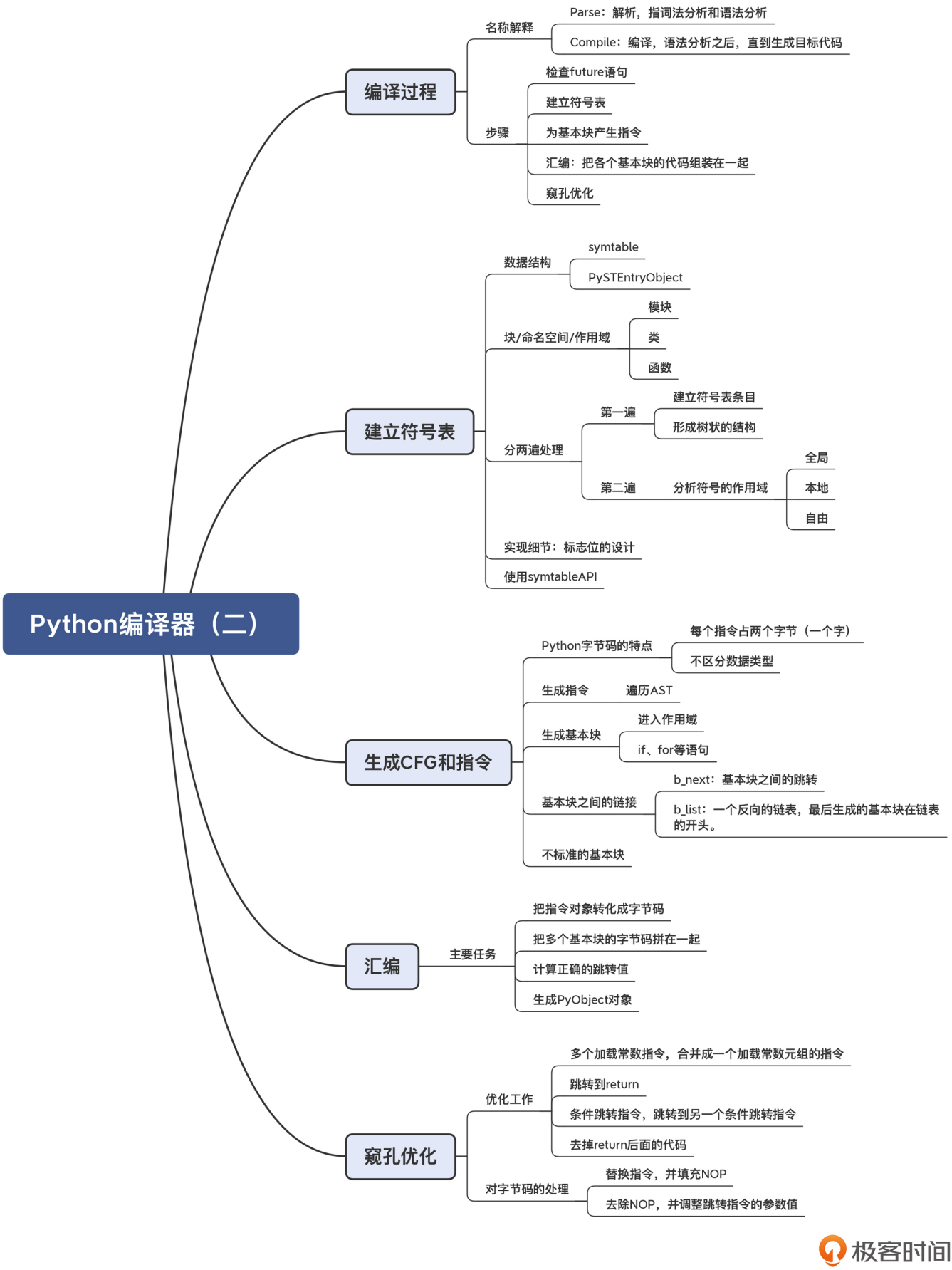
今天这一讲，我们继续深入探索 Python 的编译之旅。你需要记住以下几点：

Python 通过一个**建立符号表**的过程来做相关的语义分析，包括做引用消解和其他语义检查。由于 Python 可以不声明变量就直接使用，所以编译器要能识别出正确的“定义 - 使用”关系。

生成字节码的工作实际上包含了生成 CFG、为每个基本块生成指令，以及把指令汇编成字节码，并生成 PyCodeObject 对象的过程。

窥孔优化器在字节码的基础上做了一些优化，研究这个程序，会让你对窥孔优化的认识变得具象起来。

按照惯例，我把这一讲的思维导图也整理出来了，供你参考：



一课一思

在语义分析过程中，针对函数中的本地变量，Python 编译器没有像 Java 编译器那样，一边添加符号，一边做引用消解。这是为什么？请在留言区分享你的观点。

如果你觉得有收获，欢迎你把今天的内容分享给更多的朋友。

参考资料

Python 字节码的说明。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | Python编译器（一）：如何用工具生成编译器？

下一篇 19 | Python编译器（三）：运行时机制

精选留言（1）

写留言

Jacob.C

2020-07-13

答题：因为python变量的语义设计是“定义-使用”，不存在先使用再定义的情况，所以不用担心这种情况下的消解错误

展开